Towards Portability and Interoperability Support in Middleware for Hybrid Clouds

Ansar Rafique, Stefan Walraven, Bert Lagaisse, Tom Desair, and Wouter Joosen
iMinds-DistriNet, KU Leuven
3001 Leuven, Belgium
Email: {firstname.lastname}@cs.kuleuven.be

***Abstract*—The cloud computing paradigm promises increased flexibility and scalability for consumers and providers of software services. Service providers that exploit private cloud environments offer restricted flexibility and scalability because of the limited capacity. However, such organizations are often reluctant to migrate to public clouds because of business continuity threats and vendor lock-in. Hybrid clouds potentially combine the benefits of private and public (external) clouds. Vendor lock-in can be avoided when multiple external clouds are supported and effectively exploited.**

This paper presents a middleware platform for hybrid cloud applications. The middleware enables organizations to control the execution of their applications in hybrid cloud environments. Driven by policies, the middleware can dynamically decide which requests and tasks are executed on a particular part of the hybrid cloud. The core of the middleware, and the focus of this paper, is an abstraction layer. The abstraction layer enables portability over multiple services including data storage, blob storage, and asynchronous task execution of various PaaS platforms as well as interoperability between the PaaS platforms. We have validated the core concept by building a prototype implementation that runs on top of specific PaaS platforms as well as on a cloud-enabling middleware. A document processing SaaS application has been instantiated on the middleware. Performance results have been collected for JBoss AS cluster, Google App Engine, and Red Hat OpenShift.

***Index Terms*—Portability, Interoperability, Middleware, Hybrid Cloud, PaaS**

I. Introduction

Cloud computing is an emerging trend that refers to the delivery of computing resources as online, on-demand services, which promises increased flexibility, and scalability. The cloud computing paradigm includes three cloud service delivery models [1] (i) Infrastructure as a Service (IaaS): delivering fundamental computing resources, e.g. Amazon EC2 [2], (ii) Platform as a Service (PaaS): providing an application development and deployment platform, e.g. Google App Engine (GAE) [3], and (iii) Software as a Service (SaaS): delivering software applications as online services, e.g. Salesforce CRM [4]. PaaS can play a major role for software vendors who increasingly aim to deliver their software applications using the SaaS model. PaaS can offer a computing platform to facilitate the development and deployment of SaaS applications while simultaneously managing the availability and the scalability concerns.

However, companies are reluctant to migrate to a public cloud offering because they lose control over their applications and data as well as risk vendor lock-in. Moreover, a private cloud environment involves a large up-front investment and offers limited flexibility and scalability because of the limited capacity. In order to cope with such a situation, hybrid clouds can play an important role. The concept of hybrid cloud aims to address the trade-off between control and unlimited resources by combining a company's private cloud with one or more public cloud offerings [1]. A hybrid cloud application is typically deployed in the private infrastructure, but can burst to the public cloud to meet peak demands (i.e. spill-over). Therefore, to get the maximum benefit of unlimited resources and to keep control over application and data, companies are increasingly turning to hybrid clouds [5].

The main requirements to enable the development of hybrid cloud applications, is *portability and interoperability* across different PaaS platforms. Portability refers to the ability to port and adapt a SaaS application and its components to different platforms (on-premise as well as in the cloud) with minimal cost and effort (i.e. (re)development and training) [6], [7]. Cloud interoperability is an important property to enable the interaction between application and middleware components that are distributed over the hybrid PaaS environment. However, there exists a large variety of PaaS platforms, possibly supporting different programming languages and technologies. Even when only considering Java-based platforms, differences exist in the underpinning platform and the supporting cloud services, such as scalable storage and task queues (for asynchronous execution). Each platform offers its own vendor-specific solution for interfacing with the platform itself and its different cloud services [8], [7]. Moreover, not all of these supporting cloud services are available on every platform. This heterogeneity between the current PaaS platforms in terms of development APIs and supporting cloud services hinders portability and interoperability of SaaS applications, and thus also the development of hybrid cloud applications. For example, Google App Engine (GAE) [3] provides its own scalable datastore with a vendor-specific API while Red Hat OpenShift [9] offers support for MongoDB [10] - an open-source NoSQL database. MongoDB can thus also be deployed in a private cloud environment. Also, asynchronous processing of tasks is an explicit service available on GAE (via the Task Queue API), however, such a service is not available on OpenShift.

This paper presents a middleware prototype to experiment with hybrid cloud applications. The middleware enables organizations to control the execution of their applications in hybrid cloud environments. Driven by policies, the middleware can

978-1-4799-3360-0/14/$31.00

dynamically decide which requests and tasks are executed on a particular part of the hybrid cloud [11]. The core of the middleware is an abstraction layer that provides a uniform API for three common PaaS services, including scalable data storage, blob storage, and asynchronous task execution. This abstraction layer enables transparent distribution and interaction of the application components over the hybrid cloud. The motivation to create an abstraction layer for these three PaaS services has two reasons: (i) these PaaS services are common in typical PaaS environments, and (ii) we required these services in the selected SaaS application.

In the experiment, the middleware prototype was implemented on top of an initial set of PaaS environments consisting of two public PaaS offerings (Google App Engine [3] and Red Hat OpenShift [9]), and an on-premise cloud-enabling middleware (i.e. a JBoss AS 7 cluster). The portability and the interoperability layers were evaluated by measuring (i) the migration overhead while deploying the CloudPost application: an illustrative multi-tenant SaaS application in the domain of document processing on this hybrid PaaS, and (ii) the performance overhead introduced by this additional layer of abstraction.

The remainder of this paper is structured as follows. Section II describes the middleware architecture, with a focus on the support for portability and interoperability. Section III evaluates the migration and performance overhead of the introduced middleware layer. Section IV discusses related work. Section V concludes this paper and indicates future research directions.

II. A Middleware for Portability and Interoperability in a Hybrid PaaS

The architecture of the middleware is presented in Figure 1. It consists of an abstraction layer and a policy-driven execution layer. The abstraction layer supports interoperability in a hybrid PaaS and offers a uniform API for data storage, blob storage, and asynchronous task execution on top of heterogeneous Java-based PaaS platforms. The policy-driven execution layer enables SaaS providers to control the execution of applications in hybrid PaaS environments by means of policies. The policy-driven execution layer is described in detail in previous work [11]. Therefore, in this paper we focus on the abstraction layer. This section first describes an overview of this abstraction layer with respect to interoperability and portability support, and then focuses on the uniform API provided by the middleware for common PaaS services on heterogeneous hybrid clouds.

A. Overview of Abstraction Layer

The abstraction layer in this middleware for hybrid PaaS platforms is responsible for transparent interoperability and portability, such as data and storage abstraction. As shown in Figure 1, this layer combines three middleware components (a) the *core component*, (b) the *portability component*, and (c) the *interoperability component*. The *Core and Portability* components provide a uniform API to the application components to interact with the middleware. Each of the uniform APIs provided by these middleware components will be further discussed in the next subsection (Section II-B). The

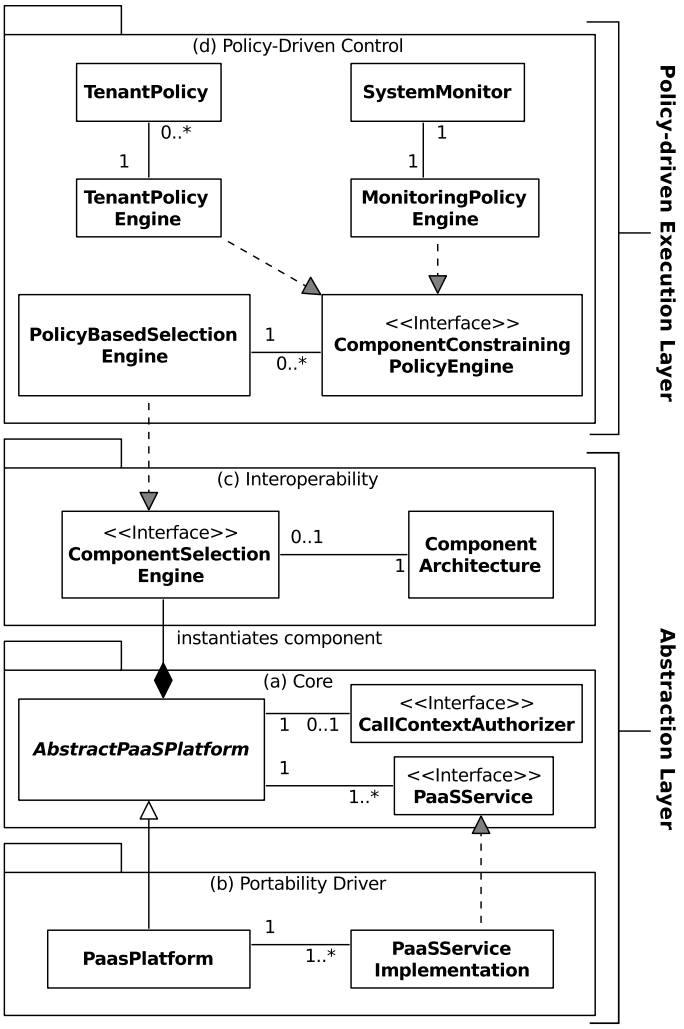

Fig. 1. Overview of the middleware architecture [11], consisting of the abstraction layer (bottom) and the policy-driven execution layer (top).

Interoperability component allows the application components to interact with the other components that are distributed over the hybrid PaaS environment. An application component interacts with the core component and requests an instance of other deployed component at the `AbstractPaaSPlatform`. The `AbstractPaaSPlatform` uses `ComponentSelectionEngine` in order to build the requested component for the application component. `ComponentSelectionEngine` initializes the requested component and after an instance is created, it is returned to the application component to invoke operations on it. The returned instance is either a local instance or a remote instance depending on the policies that apply to the provided `CallContext` object, but this is transparent to the application.

B. Uniform API for Common PaaS Services

The *core and portability* components provide a common API for structured NoSQL storage, blob storage, and asynchronous task processing. In the rest of this section, the APIs for these three common PaaS Services are discussed.

a) NoSQL Datastore API: The common uniform `DataStore` API for the NoSQL storage is presented in Figure

2. The API defines operations for the creation and retrieval of user defined keys with an optional name (String). It facilitates to create, update, and delete records based on a given key. Finally, the API also provides operations for querying and retrieving data store records. The querying happens based on the values of a partially filled `DataStoreRecord`. Each `DataStoreRecord` in a `DataStore` API is schemaless, has its own unique key, and a highly structured object that can hold any serializable value. Each key in the `DataStore` API contains tenant information to realize multi-tenancy.

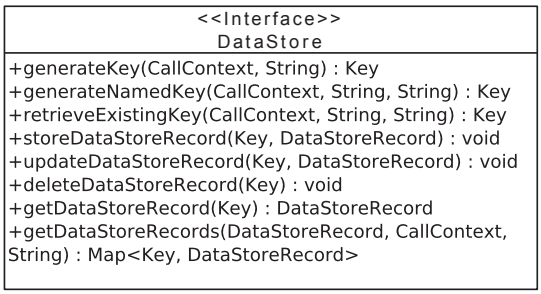

Fig. 2. NoSQL Data Store interface

b) Blob Storage API: The abstraction API for the blob storage is presented in Figure 3. The `BlobStoreImpl` API provides operations to allow application programmers to store, read, and delete blobs of data. Each blob in a `BlobStoreImpl` has a unique key and it cannot be modified once it has been stored, as the blobs are mostly relatively large. Therefore, the API does not provide any method to update blobs. Each key in the `BlobStoreImpl` contains tenant information to realize multi-tenancy. The API also provides a stream to read and write blobs. In order to ensure portability, the stream approach is translated into the underlying platform by the `BlobStore` class. This class translates the stream by writing the smaller chunks to the underlying platforms.

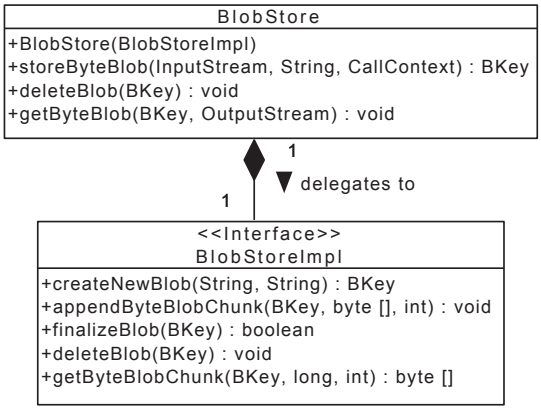

Fig. 3. Blob Store interface

c) Asynchronous Task Processing: The third service experimented with, is the asynchronous processing of tasks as shown in Figure 4. The `ShortTaskScheduler` API allows scheduling of a task in a certain queue indicated by a name (String). When a task is scheduled, it gets a unique id called task id. The task id allows application programmers to retrieve

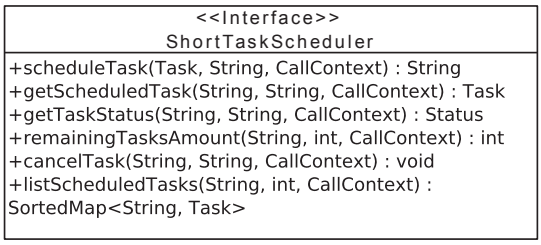

Fig. 4. Short Task Scheduler Interface

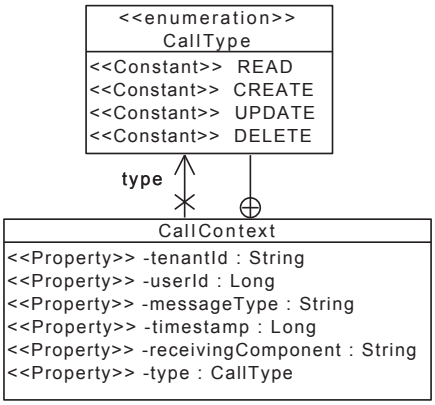

Fig. 5. Short Task Scheduler interface

the task status, get the details of the task, or cancel the task. The `ShortTaskScheduler` API also allows retrieving the amount of remaining tasks in a queue or a list of scheduled tasks. Currently, support is provided for short tasks with a maximum duration of 10 minutes. However, plans for longer processing tasks are in progress and will be addressed in subsequent work.

In all of the above services, the `CallContext` object contains all relevant information regarding the component call of a user and current tenant as shown in Figure 5. The CallContext class also supports the implementation of an application-specific authorization mechanism. The application has to implement the CallContextAuthorizer interface. The style of passing CallContext is imposed by distributed nature because, in a hybrid cloud, there is no single application run-time environment where the tenant information can be stored [11]. The information in the `CallContext` object allows the `ComponentSelectionEngine` to initialize either a local component instance, or a remote component instance. For the remote component instances, the middleware uses proxies and the remote method invocation (RMI) paradigm. The choice of the RMI paradigm is motivated for two reasons: (i) our focus is on component-based applications, and (ii) our desire is to achieve full transparency. However, the architecture is open for extensions with other communication protocols like the *simple object access protocol* (SOAP). To ensure type safety, template methods are used for requesting a component. The proposed middleware uses an *architecture description* file to construct an instance of a certain component type that is present in the mandatory `architecture.xml` configuration file loaded with the SaaS application.

III. Evaluation

The proposed middleware prototype has been evaluated using CloudPost: a document processing SaaS application. The CloudPost application is a multi-tenant SaaS application that provides online services for business-to-business (B2B) document generation. The application facilitates its tenants to generate, send, search, and archive millions of digital documents such as invoices and monthly bills on a daily basis. It is batch-driven and it consists of a set of services that are executed in a workflow. As a setup, a tenant selects one or more document templates to define the structure of the document and then upload raw data in an XML format. The raw data contains the customized content for a large set of documents and for each document it contains meta-data about the receiver and delivery method [11]. For more information about the CloudPost architecture, we refer to previous work [12].

The proposed middleware architecture is evaluated in two different ways: (a) *Migration overhead* is measured by migrating the CloudPost application to different PaaS platforms, and (b) *Performance impact* of the middleware is evaluated with or without using the abstraction layer of the proposed middleware. The purpose of the evaluation is to analyze the migration overhead introduced in the document processing SaaS application, and the performance impact of the proposed middleware. Two goals were set for this evaluation; (i) (re)implementation of only the portability driver for the desired platform in order to migrate the document processing SaaS application to heterogeneous PaaS platforms, and (ii) introduction of less performance overhead by using the proposed middleware.

A. Migration Overhead

The first aspect of the evaluation is the migration overhead. The scope of this evaluation is to minimize the migration overhead for the CloudPost application. The migration overhead is measured by migrating the document processing SaaS application to different PaaS cloud platforms. We compared the number of lines of Java code required to migrate the application for three different cloud platforms (i) An *on-premise cloud-enabling middleware* : consisting of a local JBoss 7 AS cluster with MongoDB for data storage, (ii) *Google App Engine*: consisting of App Engine with the GAE datastore, and (iii) *Red Hat OpenShift*: consisting of a Tomcat 7 gear extended with a MongoDB gear. Similarly, the total number of lines of code needed by the CloudPost application and the middleware implementation was also observed. The changes in the proposed middleware were observed for the different modules of abstraction layer and the policy-driven execution layer.

The result of the comparison is presented in Table I. It indicates that the CloudPost application stays unchanged on different PaaS platforms and only the underlying middleware changes. However, by looking at Table I in more detail, it can be observed that the *core component*, the *interoperability component*, and the *policy-driven control component* also remain unchanged over the different PaaS platforms. The *portability driver component* which provides an implementation for the different PaaS platforms only changes for different cloud deployments because each PaaS platform has its own portability driver. This also means that adding a new PaaS platform to the proposed middleware architecture only needs the *portability driver component* to be (re)implemented. This was the aspired goal while designing the proposed middleware.

TABLE I. Comparison of lines of Java code

	Local	**GAE**	**Openshift**
CloudPost	***2102***	***2102***	***2102***
Middleware Total	2374	2469	2384
↳ Core	994	994	994
↳ ***Driver***	***705***	***800***	***715***
↳ Interoperability	199	199	199
↳ Policy-driven Control	476	476	476
Total	4476	4571	4486

B. Performance Impact

A number of scenarios were run to investigate the performance impact of the proposed middleware prototype. The execution time was compared for each scenario using different cloud platforms that do or do not use the proposed middleware. The experiments were conducted on a server with two 2.40 GHz cores and 4 GB of RAM. The server used the JBoss AS cluster for the local implementation and the Google App Engine server for the Google App Engine (GAE) implementation. The choice of GAE is motivated by the presumption that on this platform the proposed middleware will have the biggest performance impact because it has the largest portability driver as shown in Table I. The comparison with the JBoss AS cluster (Local) is motivated to examine if the overhead of the middleware is consistent.

Each scenario is implemented four times: (a) two implementations without using the proposed middleware (native), one on Google App Engine (GAE) and another on the JBoss AS cluster (Local), and (b) two implementations that use the proposed middleware, one on Google App Engine (GAE+MW) and the other on the JBoss AS cluster (Local+MW). Each scenario executes the following steps in sequence (i) first, it creates a complex object which consists of a list and different type of variables, and stores the object in a *NoSQL* database, then (ii) it retrieves the stored object back from the *NoSQL* database and parses it, (iii) again, it stores a random blob of 1 MB in the blob storage, (iv) and it retrieves the stored blob again from the blob storage, (v) finally, an asynchronous PDF generation task is scheduled and the generated PDF is also stored in the blob storage. In each scenario, only the impact of the *core component* and *portability component* was considered because the *interoperability component* and *policy-driven control component* do not have native counterparts. For each platform, 4 performance experiments were executed. For each experiment, we executed the scenario 2000 times in 5 runs of 400 executions. The first execution of each run was always ignored because it involves different PaaS services to be started, which might have influenced the measurements. The first experiment measured the execution time of the complete scenario for the native implementation. The second experiment is measured with an implementation that uses the proposed middleware. The third experiment measured the execution time of the individual scenario steps for the native implementation. The fourth experiment measured the execution time of

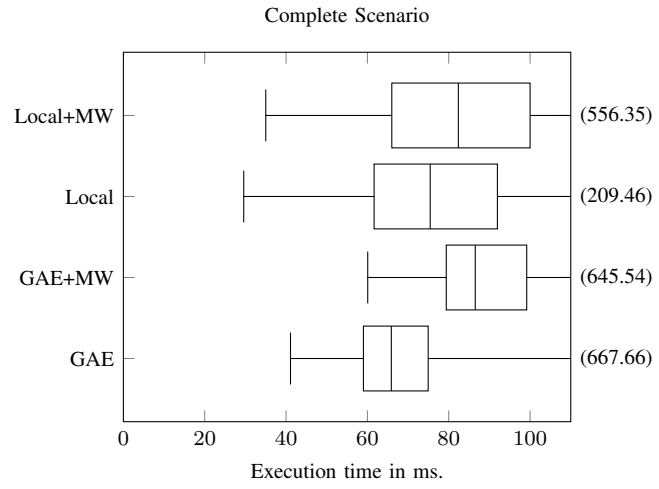

Fig. 6. Box plot of measured scenario execution time

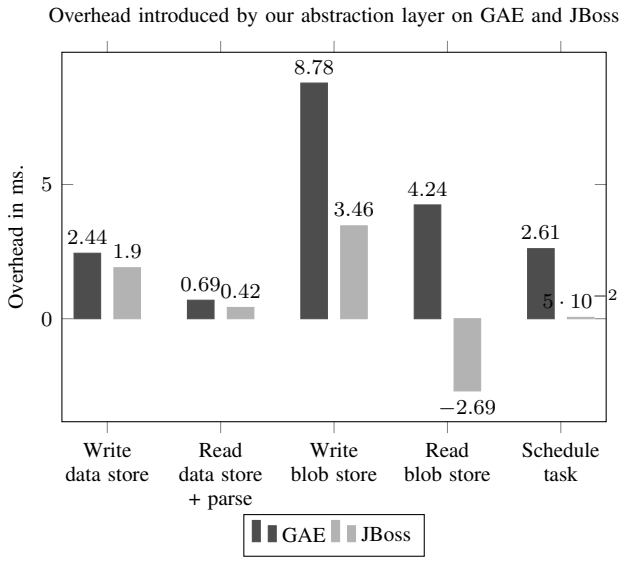

Fig. 7. Overhead introduced by our abstraction layer for the individual cloud service operations on GAE and JBoss

the individual scenario steps using the proposed middleware implementation. After executing more than 400 scenarios in a single run, it was observed that the data management and persistence overhead by the PaaS platform became too big to measure the middleware impact. Therefore, after each run, the development server was restarted to clear the data cache. The same evaluation might not be optimal on Google App Engine (GAE) because a lot of other factors can influence the result: e.g. automatic scaling factors as well as a load of other users of this public cloud.

TABLE II. MEDIAN OF SCENARIO STEPS AND COMPLETE SCENARIO EXECUTION TIME IN MS.

	GAE	**GAE+MW**	**Local**	**Local+MW**
Write data store	1.11	3.54	0.30	2.20
Read data store	0.44	1.12	0.54	0.96
Write blob store	39.14	47.91	42.31	45.78
Read blob store	4.44	8.68	4.03	1.34
Schedule task	0.57	3.18	0.01	0.06
Total	65.88	86.52	75.44	82.36

The results of these experiments are presented in Table II. The proposed middleware introduces a performance penalty of 20 ms for Google App Engine (GAE) and 7 ms for the JBoss AS cluster (Local) by looking at the overall execution time of the test scenarios as illustrated in Figure 6. Figure 7 shows a more in-depth look of the middleware abstractions. It is clear that mainly the write operation on the data store and the blob store suffer from extra overhead. The overhead introduced by the proposed middleware has two main reasons: (a) it adds an extra layer of indirection which inherently introduces extra overhead, and (b) it has to combine several PaaS services to be able to offer a uniform PaaS interface. For example, multi-tenancy causes an overhead when the meta-information about the current tenant is stored in data objects and blob objects. This meta-information is important and required for data isolation. Before reading or writing any data with a certain key, the proposed middleware always checks if the key belongs to the requesting tenant. This ensures data-isolation, but always requires extra data store access introducing additional overhead.

The reason that the proposed middleware reads blobs more efficiently than a native JBoss implementation, can only be due to the fact that the middleware uses the `ReadableByteChannel` Java API and the native implementation uses the `InputStream` Java API.

IV. RELATED WORK

In previous work [7], we investigated current PaaS offerings in light of SaaS development based on a practical case study. The work identifies three categories of PaaS platforms, with different levels of support for portability. However, the results of this comparison clearly indicate the need for standardization to improve portability of SaaS applications across different PaaS platforms and to tackle vendor lock-in, especially with respect to typical PaaS services such as scalable storage.

Cunha et al. [13] proposed a middleware architecture that facilitates dynamic deployment, registration and portability of services, and enables developers to create and expose services using a cloud based service delivery platform within a service-oriented architecture (SOA). Although, they also focused on portability and application migration, there are still significant differences with our research. The key differences are (i) their focus on SOA applications differs from ours, which is more on component-based applications, and (ii) the authors acknowledged the need for a uniform PaaS API, whereas we have implemented an initial common PaaS API for three heterogeneous hybrid PaaS platforms.

The research conducted by Mietzner et al. [14] focuses on cloud application portability. The authors proposed an extension to the service component architecture (SCA) with variability descriptors and multi-tenancy patterns. However, SCA applications can only be executed in an SCA application environment, whereas our research focuses on platforms that do not support SCA. Moreover, their research focuses on multi-tenant customization, packaging, and deployment migration obstacles while our research addresses data isolation and

implementation of a common PaaS API for hybrid PaaS platforms. Paraiso et al. [15] is an example of a PaaS infrastructure that allows service component architecture (SCA) applications to deploy on heterogeneous PaaS clouds. However, their infrastructure relies on their own FraSCAti application environment, whereas, we focus on infrastructure that does not support SCA. In addition to this, we address multi-tenancy while they do not consider this explicitly.

Another related solution is provided by the European mOSAIC project [16], where an independent PaaS platform API is developed to provide support for heterogeneous hybrid clouds. The developed API uses a driver architecture and can be deployed on top of heterogeneous hybrid PaaS platforms. Our approach is similar to their approach, but with different focus. In their research work, they have focused on a PaaS API for task automation whereas our research focuses on an abstraction API for storage services.

Another design of PaaS is presented in DRACO [17]. The authors proposed a new PaaS platform that is inspired by FCAPS, the ISO telecommunications management network model. The proposed PaaS platform is built on top of an IaaS layer and can be utilized by other SaaS applications or PaaS platforms. The focus of DRACO is (i) to address issues concerning PaaS management such as fault tolerance, configuration, accounting, performance, and security, and (ii) to provide a platform for the development of algorithms that require parallel processing and a considerable amount of computation in the cloud. In contrast, we solve application-level issues and provide portability and interoperability support across existing PaaS environments.

V. Conclusion and Future Work

Support for hybrid clouds has the potential to offer open deployment environments that avoid vendor lock-in and that reduce the risk of investing in potentially obsolete technologies. However, the available support for hybrid cloud applications currently is very limited. This paper proposes a basic middleware architecture that offers enhanced support for hybrid cloud applications by defining a common API for a couple of core services in typical PaaS platforms. The API supports portability and interoperability for storage and for managing asynchronous tasks, thus increasing control for application stakeholders. The middleware is aimed at PaaS platforms, but indirectly, it also supports IaaS clouds through the use of cloud-enabling middleware, such as current application servers.

We have evaluated the overhead caused by our protoype implementation and we have measured the performance penalty when depolying an illustrative SaaS application in the domain of document processing. In particular, we have demonstrated that our middleware offers benefits such as application portability and interoperability. In contrast, a SaaS provider clearly has to make the trade-off between the performance overhead of a native implementation versus the benefits offered by an extra middleware layer.

Clearly, portability and interoperability in the context of hybrid clouds remain important challenges. Further validation and improvement of the proposed abstraction layer is required by supporting more storage services, more PaaS platforms, and by testing and evaluating the cost and benefits for other types of SaaS applications. In future work, we plan to extend our middleware to support other PaaS services, such as addressing long-running jobs. Finally, another line of work we want to focus on is (i) to optimize the current driver implementations, for example to minimize the overhead introduced by Google App Engine on the datastore and blob store (see Table II), and (ii) to minimize the performance overhead through more efficient use of memory caching.

VI. Acknowledgments

This research is funded by the iMinds DMS2 project. The iMinds DMS2 is a project co-funded by iMinds, a research institute founded by the Flemish Government. Companies involved in the project are Agfa Healthcare, Luciad, UP-nxt, and Verizon Terremark.

References

[1] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," http://collaborate.nist.gov/twiki-cloud-computing/pub/CloudComputing/Documents/Draft-SP-800-145_cloud-definition.pdf, Special Publication 800-145, 2011.

[2] Amazon Web Services LLC, "Amazon Elastic Compute Cloud (Amazon EC2)," http://aws.amazon.com/ec2/, [Visited on 20 December 2013].

[3] Google, Inc., "Google App Engine," http://code.google.com/appengine/, [Last visited on 20 December 2013].

[4] Salesforce.com, Inc., "Salesforce CRM," http://www.salesforce.com/, [Last visited on 20 December 2013].

[5] N. Leavitt, "Hybrid Clouds Move to the Forefront," *Computer*, vol. 46, no. 5, pp. 15–18, 2013.

[6] K. Garen, "Software Portability: Weighing Options, Making Choices," *The CPA Journal*, vol. 77, no. 11, pp. 10–12, 2007.

[7] S. Walraven, E. Truyen, and W. Joosen, "Comparing PaaS offerings in light of SaaS development," *Computing*, pp. 1–56, 2013.

[8] D. Petcu, "Portability and Interoperability between Clouds: Challenges and Case Study," in *ServiceWave '11: Towards a Service-Based Internet*. Springer Berlin Heidelberg, 2011, pp. 62–74.

[9] Red Hat, Inc., "Red Hat OpenShift," https://www.openshift.com/, [Last visited on 20 December 2013].

[10] MongoDB, Inc., "MongoDB," http://www.mongodb.org/, [Last visited on 20 December 2013].

[11] T. Desair, W. Joosen, B. Lagaisse, A. Rafique, and S. Walraven, "Policy-driven Middleware for Heterogeneous, Hybrid Cloud Platforms," in *ARM '13: Proceedings of the 12th International Workshop on Adaptive and Reflective Middleware*. NY, USA: ACM, 2013, pp. 2–1.

[12] W. Joosen, B. Lagaisse, E. Truyen, and K. Handekyn, "Towards application driven security dashboards in future middleware," *Journal of Internet Services and Applications*, pp. 2–4, 2011.

[13] D. Cunha, P. Neves, and P. N. M. d. Sousa, "Interoperability and Portability of Cloud Service Enablers in a PaaS Environment," in *CLOSER '12: Proceedings of the 2nd International Conference on Cloud Computing and Services Science*. SciTePress, 2012.

[14] R. Mietzner, F. Leymann, and M. P. Papazoglou, "Defining Composite Configurable SaaS Application Packages Using SCA, Variability Descriptors and Multi-tenancy Patterns," in *ICIW '08: Third International Conference on Internet and Web Applications and Services*, June 2008.

[15] F. Paraiso, N. Haderer, P. Merle, R. Rouvoy, and L. Seinturier, "A Federated Multi-cloud PaaS Infrastructure," in *CLOUD '12: IEEE 5th International Conference on Cloud Computing*, June 2012, pp. 392–399.

[16] D. Petcu, G. Macariu, S. Panica, and C. Crăciun, "Portable Cloud applications - From theory to practice ," *Future Generation Computer Systems* , vol. 29, no. 6, pp. 1417–1430, 2013.

[17] A. Celesti, N. Peditto, F. Verboso, M. Villari, and A. Puliafito, "DRACO PaaS: a Distributed Resilent Adaptable Cloud Oriented Platform," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, May 2013, pp. 1490–1497.